

Animation Blending: Achieving Inverse Kinematics and More

Jerry Edsall

July 4, 2003

1 Introduction

Traditional Inverse Kinematics (IK) systems attempt to solve a fundamental artistic problem with math. While the math may technically bring a character's leg to the correct position, it rarely imparts a sense of weight or timing. This article presents a technique for creating IK effects without these problems. The method uses only artist-created animation that is blended to perform IK. By using animation only, the artist retains control of the entire process. The character can now struggle as he runs uphill while shifting his weight and speed, all without losing accuracy of the foot positions which match the grade of the ground. This article illustrates the practical applications of multi-axis blending, blend hierarchies and priority blend trees as well as the pitfalls of using such a system in the real world.

Figure 1: MechWarrior



The fundamental technique of blending animations together is not new. The traditional blending approach consists of transitioning from one anima-

tion to another seamlessly. Creating transitions exercises only a small amount of what can be achieved with blending. Blending can provide robust IK solutions, complicated dynamics, as well as other feedback effects, and all of these can be accomplished simultaneously. Using blending to perform IK effects allows a much more stable solution than using traditional math-based IK solutions. A traditional IK system will simply solve a chain of joints to a specific goal position. The IK systems completely overlook the weight and dynamics of the object. This can result in uninteresting animation that is technically accurate but lifeless and twitchy.

The basic concept of single-axis continuous blending is widely used in games. The techniques presented here extend this concept with the introduction of multi-axis blending, blend trees and priority blend trees. This combination of techniques permits complex layered animations. Multi-axis blending simultaneously allows the developer to maintain IK accuracy and the animator to create the motion. When the artist is in charge, the resulting character-motions embody a sense of weight and timing that are unobtainable through pure mathematic formulas. Beyond IK, the developer can determine all manners of feedback systems, such as taking damage and performing fight moves. A priority blend tree allows for layering of blend effects with more traditional transition animations and procedural animation.

2 The Blending Decision

Why chose blending? IK solutions often suffer from unpredictable results. When IK solutions are either under- or over-constrained, the resulting solution

can produce strange behavior. Many animators are familiar with having to move an IK handle around when suddenly the chain rotates 180 degrees and bends backward unpredictably. Even high-end 3D packages suffer from these issues. This problem worsens the longer and more complicated the chain becomes. In addition to generating IK effects, it is also desirable to have other feedback effects as well, such as blended transitions. Having the core animation engine written specifically to handle large scale multilayer blending makes these tasks easier. The animation system being presented is art driven and artist controlled. These are the major reason why the Fasa Studio MechWarrior team uses a blended system. The MechWarrior team's goal is to impart the enormous size and weight of the giant walking robots (referred to as "'Mechs") at all times. 'Mechs weigh 100 tons and are as tall as buildings. The player needs to feel this size and power every time they see a 'Mech.

Blending systems are not without cost. There are significant downsides to large scale blending. Data size, large content requirements, CPU performance and memory requirements all make this method challenging. Blending systems can require very large data sets. To obtain high-quality results, multiple base animations are necessary. Each 'Mech is comprised of 152 animations. At any time, each 'Mech can be playing up to 15 simultaneous animations. In a 24-player game, the working set of animations adds up quickly. CPU and memory performance can also contribute to poor performance, so a close watch needs to be applied to both.

It is important to remember that blended IK solutions are only approximate blended solutions. This is the exact opposite problem posed by traditional IK. When using IK, the character's feet will touch the ground, but can experience minor sliding, foot penetration or even floating. These effects are only minor and can be improved in a number of ways. A blending system can be augmented with procedural animation, traditional IK and physics. With blending, one can interlace procedural animation before, during and after the blend. The MechWarrior engine procedurally animates torso rotations, foot rotations and even dampens out joints.

Given two animations that represent two posi-

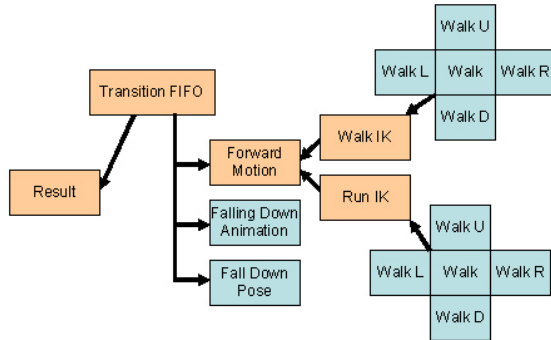
tions in a given space, interpolating from one animation to the other will represent any position between those animations. This is the basic theory of performing IK and other feedback effects with blending. These animations can be static poses or motions over time. Each animation represents a target motion or pose. For blended IK, the animations can represent walking uphill and downhill, or grabbing a cup at 2 meters and grabbing a cup at 3 meters. Other feedback effects could represent turning on a snowboard or damage taken to a character. Each animation helps define the available target space. It is possible to add any number of additional axes to the blend space. It is also possible to add more animation to different points within the existing space to help define that space more accurately. For instance, combining an animation of a character walking uphill and downhill may not necessarily result in an animation of the character walking on flat ground. In this case, one would further define the space with an animation of walking on flat ground. With these three animations, the blending space is now fully defined.

3 Overview Of The MechWarrior engine

The MechWarrior engine is comprised of two main systems: the animation playback system and the blender system. Animation playback is a generic animation loader and player. The blend manager handles the hierarchical blending using blender nodes which each represents a hierarchical layer of the blend tree.

Figure 2 is a snapshot of a MechWarrior blend tree. Each frame in the blend tree is adjusted and the weights are calculated. This diagram depicts blender nodes that determine the position of the IK. The IK blender nodes calculate the target IK position. The result of the IK node feeds into the forward motion blender. By blending a walking animation and a running animation, the forward motion blender can represent any speed between the two source animations. The resulting jog animation is then feed into a transition engine that

Figure 2



is altering the movement state of the ‘Mech from a jog to falling over animation. Each of these blender nodes will be covered in detail starting with a quick overview of the lowest level animation player and working up to the blender algorithms and blender nodes. Then the presentation will cover how to put all the pieces together into a final system. Other topics include performance a discussion of the implications that this system has on art staff and tools that can be written to improve their processes.

4 Animation Player

The MechWarrior engine uses quaternions for rotations. The basics of interpolation will not be addressed in this presentation. The following are some good references on Quaternion math. It is recommended that an optimized spherical lerp be used, as a traditional spherical lerp can get quite expensive.

The lowest level of the animation system handles generic animation playback. The player does not know anything about blending, only animation loading, iterating and key frame interpolating. A single static pose can represent an animation just as motion over time. All the blending techniques work equally as well with a simple pose, so it is important to consider this alternative for memory

and CPU consumption. The animation format supports vertex, velocity, position and rotation. The interpolator supports linear, spherical, optimized spherical and snap interpolations.

Animation time can be advanced, decremented and set by frame, percent and seconds. Time advance of the animations happens independently of the interpolation. This is a performance and memory optimization. The blend manager has the ability to cull out the interpolation of animations that may be insignificant but does not know this until the final blend tree is created.

The velocity of the character is encoded into the animation from the art package. By encoding the velocity into the animation, the blend manager can also blend velocities. This important technique allows the animator more freedom in how he animates. For example, the animator can animate in an arbitrary space for each animation but is not required to keep a constant speed during an animation. When the simulation is notified of the final joint positions, it is also notified of the velocities. It is up to the simulation to either ignore or apply this velocity.

5 The Blend Manager

The blend manager is comprised of a tree of blender nodes. Each blender node holds some number of other blender nodes. A transition blender node, for example, holds two other blender nodes; the blend node it uses for the start animation and the blend node for the end animation. Base animations are also represented by a blender node. Abstracting the blend into a hierarchical list affords the programmer some luxuries in code. It becomes very complex to generate meaningful blend weights when dealing with more than a few blend values in a flat list. When using a hierarchy the code makes localized decisions about the appropriate weight. The code, for instance, will only determine the necessary weight for the walk and the run, as well as the transition weights for the forward and fall motions. The blend manager then flattens the list and normalizes the weights for final interpolation. The

blend manager can also remove duplicates and cull insignificant branches of the blend tree to improve the performance of the final interpolation.

Figure 3

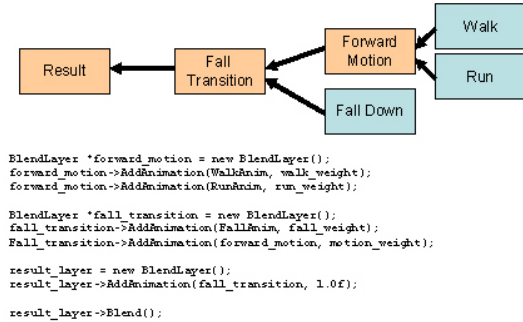


Figure 3 represents a character that is moving forward, which is a combination of a walk and run in a continuous blend. The character may be at a jog speed in this diagram. But, the character is also in the process of falling down. The jogging animation is still being calculated during the course of the transition blend so that the result remains smooth. The falling down animation could further be the result of a blend, as could the walk and run. This illustrates a straightforward use of the blending engine. Later, different encapsulations of blend layers will be discussed that further simplify use.

The process of blending includes four steps. The first step is to build the tree, which the simulation does based on the movement states of the character. The next step is to advance the time on each animation in the tree. Third, the list is flattened and all the weights are unified. The last step is to interpolate the current position of each source animation and then blend that value into the final result buffer. At this point one can further blend with procedural animation. This can be as simple as pointing a head or complex IK chains. It is also possible to add a blender node with procedural data if the desire is to have it blend with the rest of the system.

Figure 4

```

Flatten( BlenderNode *current_node, float current_weight, BlenderList flat_list )
{
    current_node->blendWeight = current_weight;
    if (current_node->IsAnimation())
        flat_list.Add(current_node);
    float total_node_weight = 0.0f;
    for(int i = 0; i < current_node->childCount; i++){
        total_node_weight += current_node->children[i]->blendWeight;
    }

    for(int i = 0; i < current_node->childCount; i++){
        float child_node_weight = 0.0f;
        if (total_node_weight > 0.0f){
            child_node_weight =
                current_node->children[i]->blendWeight / total_node_weight;
        }
        child_node_weight *= current_weight;
        Flatten(current_node->children[i], child_node_weight );
    }
}

```

The basic algorithm for unitizing the weights in the blend tree is presented in Figure 4. The algorithm recursively steps through each layer of the blend tree. This builds a flat list of the animations in the tree while unitizing the weights. Each animation will now have its own percentage relative to all of the other animations. All there is left to do is the actual blending.

Figure 5

```

Blend( ResultBuffer *buffer, BlenderList flat_list )
{
    float current_weight = 0.0f;
    for(int i = 0; i < flat_list->childCount; i++){
        BlenderNode *node = flat_list->children[i];
        if (node->blendWeight > 0.0f){
            current_weight += node->blendWeight;
            float interpolation = node->blendWeight / current_weight;
            BlendChannels(buffer, node, interpolation);
        }
    }

    BlendChannels(ResultBuffer *buffer, BlenderNode *node, float interpolation)
    {
        For Each Channel
            interpolate current value from time
            interpolate from buffer to current value by interpolation
    }
}

```

The blend algorithm that the MechWarrior engine is pretty straightforward. See Figure 5. The algorithm interpolates between each animation based on the running percentage. For linear translation, this method will provide exact results. For rotations, this method is only approximate. While there are other methods for adding quaternion rotations together, this method is the simplest in preserving the spherical nature of the rotations being

blended. The first part of the algorithm calculates the current interpolation value to be used. Then it calls down to the animation player to actually calculate each animation channel. Each channel will interpolate between the appropriate key-frames based on time, and then, the outcome will be interpolated into the result buffer by the unitized blend percentage.

6 Cross Fade, Continuous Blend and Feather Blending

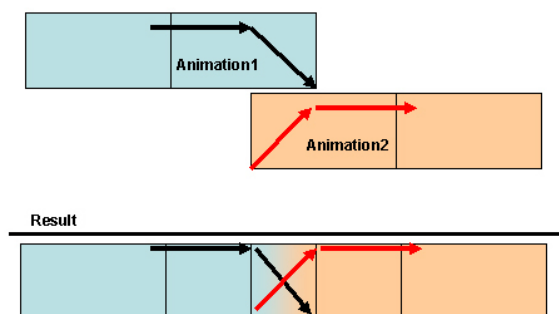
Cross fades and continuous blends are basic ways of using the blender engine. Figure 6 represents the flow of a simple cross fade transition. The upper blue box represents animation 1, and the beige box is animation 2. The arrows represent the blend weight of each animation. In this example, the two animations are cross faded with linear weights to form a transition. When the game decides to transition from one animation to another it simply increases the weight of animation 2 while simultaneously decreasing the weight of animation 1. The end result is, hopefully, a seamless transition.

Figure 7 illustrates a continuous blend. Both animation 1 and 2 are cycling indefinitely. In this case, animation 1 represents an undamaged walk, and animation 2 is a damaged walk. At some point, the game decides that the walk should now be damaged to some degree. The simulation increases the blend weight of the damaged walk, and the outcome will be a mixture of the two animations.

The purpose of feather blending is to overlay an animation without having all the joints blend into the destination at a unified weight. Feather blending is just an extension to the overall blend weight of transitions and continuous blends. An example would be an animation of a character running and a separate animation of the character standing still while firing a weapon. By putting modifiers on the blend weight at each joint level, you can blend the two together to have the character run and shoot. Each joint would specify how much of the over-

all blend weight it would inherit. This allows the shooting animation to not affect the running animation. Without the feather blend, the animation would perform a strange stand/walk mixture that would look incorrect. Each channel adds a modifier to the overall blend weight of the animation.

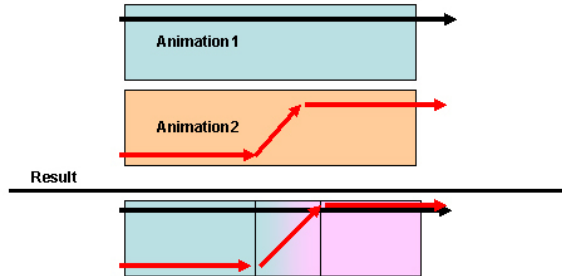
Figure 6



Creating well-blended transitions is not as easy as placing two animations into a blending engine. A bit of care is required to make satisfactory transitions and continuous blends. When transitioning from animations that are not similar, such as a character walking and a character swimming, the resulting animation will very likely look erroneous. Animations that are not similar tend to have strange continuous blends. Clearly, the closer the animations are to each other, the more seamless and natural the blend will appear.

For radical blends, the minimum requirement is at least one point in each animation that is similar. The transition can be performed at this similar point. This issue is most prevalent with leg cadence, which is the speed in time that legs repeat their cycle. The animator will need to precisely pick transition times to match leg cadence. If the cadence is not aligned, the resulting motion is legs that cease their pendulum motion. The legs may pause, stutter or travel backwards. These timings may need to be per transition, as each transition destination could have a differing leg cadence. This is where there was a lot of trouble with MechWarrior. In the end, the artists needed to look at every

Figure 7



transition to make sure that the leg cadence was consistent.

While an animation may be similar, its playback rate may not be, such as a walk or a run. Both animations can start on the left foot and complete a full cycle of walking motion, but each animation has a different run length. The walk could be completed in one second, while the run could be twice as fast. When blending these animations together continuously, strange results will occur as the run will be played twice for each cycle of the walk. This time, the cadence is not the same rate, even though the cadence has the same start point. The result of this problem will likely be an interesting skipping animation, but probably not the desired result. To combat this problem, simply scale the playback time of the animations in proportion to the weight. Now, when the animations are played, each animation will cycle at the same playback speed. For the walk/run example, when the walk is being played at full speed the run would be played at half speed, so the cycle rate is the same. As the walk transitions to the run animation, the walk playback rate will increase to double speed at the end of the transition. To prevent abrupt changes in momentum, it is essential to alter the play back rate based on the blend value. Matching playback rate is less of an issue for transitional blends, but the problem could still arise if the transition is of significant length.

Animation transitions that are simple cuts should still have some overlap. This was a tragic

mistake made in 'Mech that resulted in countless hours of tracking down pops and glitches in the animation by both the art staff and programming staff. The hand-animated transitions had matching edges on the animations with no overlap. This cut technique was same as was done with previous games, but proved to be ineffective for a blending system. The blending invariably results in mismatched animations with pops that are not fixable. A tiny amount of space before and after each animation would have fixed this, providing the necessary space to blend out. The newest revision of the animation system utilizes one second of header and trailer on each transitional animation which allows ample space for blends. From an artist's perspective, it is significantly easier to have this data in the file upfront rather than having to add the data later.

7 Blender Nodes

Figure 8

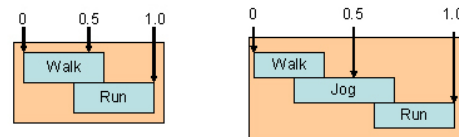
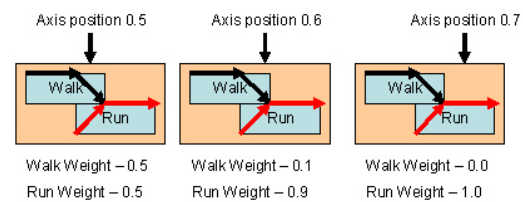
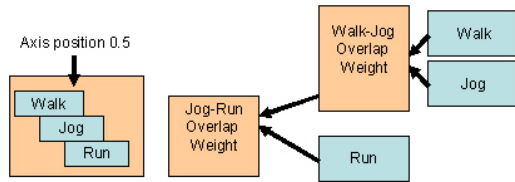


Figure 9



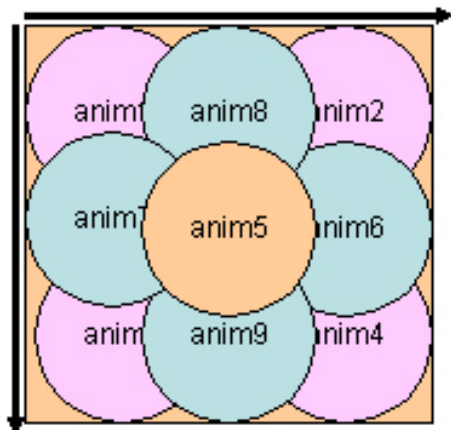
A single axis blender allows you to define multiple animations that will be continuously blended. A range is defined for each animation in the blender node, which specifies the location on the axis where the animation will have a blend weight. As the axis is moved by the simulation input, the blender cal-

Figure 10



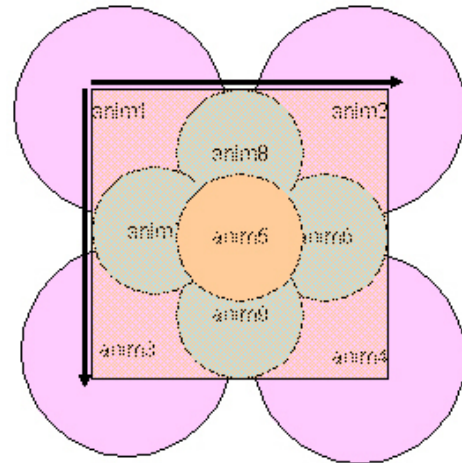
calculates the weight of the animations. In the case of Figure 8, the input comes from speed. The weight of the overlapping animation is linear, based on the current position in the overlap. In the illustration, the blend weight is linear inside the overlapping area of the animations. The blender node in Figure 9, given a point in the defined space, will output blend weights that are appropriate. The combination of multiple animations active in the same axis blend are simplified by using blend layers to resolve the weight (See Figure 10). This makes the solution much easier, since the animation ranges are arbitrarily complex. The only limitation to the layer solution is the need to keep each successive animation starting at or above the last animation axis start position.

Figure 11



Multi-axis blenders allow users to blend any number of axis (See Figure 11). This is the cornerstone of producing IK effects. Each animation is given a sphere to represent its weight in the space, and each sphere has a center point and a radius. The center point of each circle represents the animation at full weight all the way to the edge of the circle, which is zero weight. When the blender node calculates the weight it steps thru each animation, calculating the weight by its distance to the input. The calculation caps the weight at the range of the sphere. For IK effects, each animation represents a target IK position, be it a ground angle to walk on, or a point in space to grab. Notice that the defined space in Figure 11 has some gaps. If those gaps are left in, the blender node will not assign weight to anything and the animation will not appear correctly. This illustrates the biggest challenge of using this type of blender. The easiest way to deal with this issue is to increase the radius of effect for specific animation (See Figure 12).

Figure 12



Input values feed into the blender may not always be smooth because they come from the simulation. They may be damage, speed or hill grade, which may or may not be consistent from frame to frame. The MechWarrior engine handles this by adding a set of generic smoother classes to filter input with. How the number is smoothed varies depending on what the animation is driving. Currently, support for acceleration drag models, springs and simple

averages over time are used. All are useful methods for preventing unsightly pops.

8 Trees and Lists

There are a few generic tree and list structures that will assist in further abstracting the blend hierarchy. Trees and lists are still blender nodes, and as such, can be children and have children. They each represent different generic ways of calculating blend weights of their respective children. Lists assist in maintaining important animations visible while also keeping the blending engine from blending too much data. If too many animations are blended at once, not only will the performance drop, but the resulting animation will become muddled.

Priority trees guarantee that certain animations are played regardless of other animations blend weights. Starting with the top level, each blender node gets as much weight as is currently available and requested by the blender node. The next layer of blender nodes receive the remaining weight and so on. The following example illustrates this point. In a priority tree of three animations, the first two request seventy 70% of the weight each. The third request 100%. The first animation is granted the full weight of 70%. 30% of the weight remains to be disbursed. The second animation receives 70% of the remaining 30%, or 21%. The 9% remaining is then granted to the third animation. So, if the second animation had requested 100%, there would have been no weight remaining to disburse to animation three.

Overlay lists combine animations that are feather blended. These are very similar to single-axis blenders, except overlay lists do not unitize the weight of their children. The weight calculation and blend algorithm need to be extended to support overlays. Without extending the algorithms they would incorrectly unitize the base weights of the feather blends.

A FIFO priority tree allows animations to gain and lose priority based on when they are queued. This could include a selection of hit effects or spe-

cial move animations. No particular move is more important than the last, but it is desirable to have each animation interrupt the last. This is where a transition engine would be placed to have specific transition events and information. A FIFO tree can calculate weight the same as the priority tree, or it can have a more complex transitional engine built on it to calculate weights more specifically. The MechWarrior engine uses a complex FIFO tree that manages specific transitions from each animation. It manages all in-out time marks for each animation as well as motion state information for the simulation.

9 Putting It All Together

Large selections of blending methods have been defined and are almost ready for use. When using a blending system, the level of required animation is significant. It is critical to plan out the entire blend tree from the start. When planning the 'Mech animation system, our team was surprised by the amount of animations it took to fulfill the design. MechWarrior required 152 animations per 'Mech, and there were over 20 'Mechs. During scheduling, it became evident that generating this amount of animations was not obtainable. Because of this, the team was forced to modify the animators' tools to minimize the amount of animations. Ultimately, the list was reduced to 30 animations that the animators created from scratch. At that point, tool processes would take those 30 animations and alter them in various ways to create the final 152. The animators then performed a final review of all 152 animations before they were added to the system.

Testing and finalizing each blender is crucial. In developing MechWarrior, the team failed to test a few blocks, such as 'Mech jumping. There was a general plan, but nothing concrete. Unfortunately this became a bigger problem later in the project. By the time it was obvious that there was a real problem, half the 'Mechs were already animated and there was not enough time to redo the animations or reschedule the remaining 'Mechs. The final result was that the game was shipped with unsatisfactory landing animations. This would have been

solved if time were taken upfront to make sure all the animation blocks worked as planned.

Figure 13

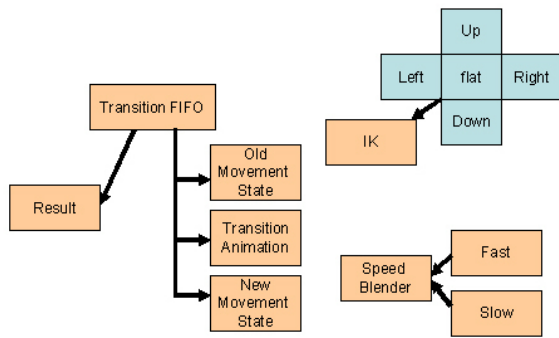
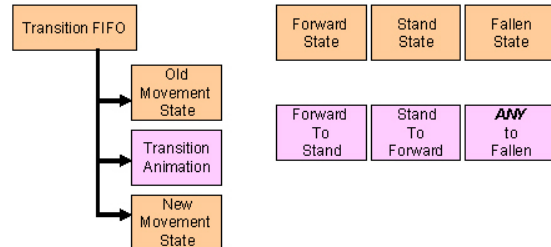


Figure 13 represents the basic blenders used for MechWarrior. 'Mechs are animated with a state engine which is also serves as the transition FIFO. Each state represents a cycling animation like walking, standing or fallen down and is built off of other blender nodes. Each transition is a one-time animation. The transition FIFO is extended to have knowledge of each state and is responsible for pushing new animations into the queue based on input from the 'Mech simulation. As it pushes new states, it also handles pushing the transition blender nodes as well. The FIFO carefully monitors positions of all states to handle the addition and removal as well as the transition points.

Some of the states represented in 'Mech are simple poses, while some are multi-axis IK blender nodes, like crouching or standing. Other states are built up of single axis speed blenders and multi-axis IK blenders. Forward motion, for instance, is a speed blender with a walk input and a run input. Both the walk and run inputs are multi-axis IK blenders with up, down, left, right and flat animations feeding into them.

The transition FIFO is also the state engine for the 'Mech animations. The states are code driven. The 'Mech simulation code requests different motions states depending on need. The contents of the states and transitions are data driven from a

Figure 14



file. It is not the responsibility of the simulation to determine what is in a state. The simulation simply requests states and waits for them to happen, but must also be aware that the state change will not be automatic. Each state and transition is represented by some unknown blender nodes. The blender nodes could be as simple as poses or as complex as full trees. In this case, the FIFO is relying on states to be cyclic and on transitions to be non-cyclic. When the simulation requests a state, the FIFO finds the state and the associated transition. The transitions could be specific, such as a specific forward state to stand state, or they could be generic, like any state to fallen state (see Figure 14). The transition is associated with transition points for the old and new states. The FIFO waits for the old movement state to be in the correct position and then starts the transition blends. When the old state no longer has any influence, the FIFO removes it from the queue. When the transition has no more influence, it is also removed from the queue, leaving just the new movement state. At this point, the simulation is ready to transition to another node. Theoretically, these nodes can be stacked in multiples and processed one at a time, but that could grow out of control and take an inordinate amount of time. The MechWarrior engine is careful to request only one state at a time. In the rare case that a state change really needs to happen instantly, the simulation does a special request which puts a new priority blender node on top of the transition FIFO. When the new animation gains full weight, all others are removed and that animation becomes the current movement state.

Figure 15

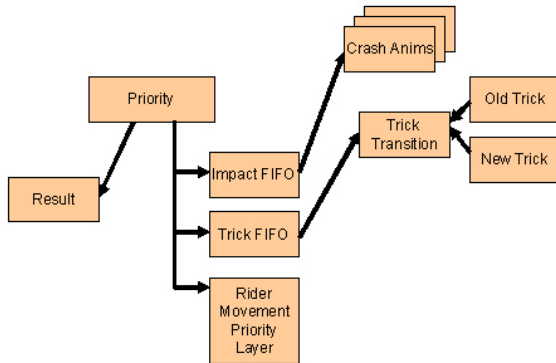
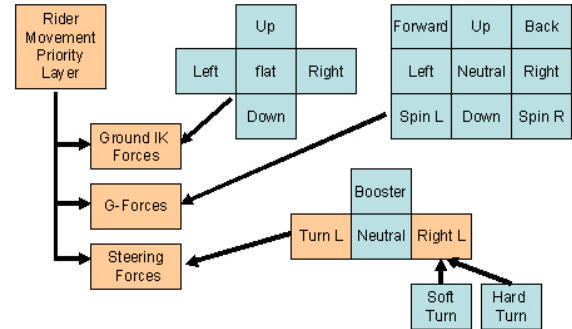


Figure 15 is a tree of an action-sport title currently being developed. The tree is much more complicated than the 'Mech tree, and this tree relies much less on states. In this tree, there is a top-level priority system to determine what general blender branch will be shown. The character in this example can do tricks, ride his vehicle and crash. Crashing is the most important and should always be visible. There are numerous ways the rider can crash but only one should play at a time. As such, this branch is a simple FIFO queue. The next branch is the trick branch. Like the crashes, only one animation is played at a time, but the transitions are a little more important and defined. Tricks have a transition state engine that is also a FIFO queue similar to the 'Mech state engine. The last layer of the priority FIFO is the movement layer. The movement layer is again another priority layer. However, blending too many animations look muddled. The priority systems assist the simulation in culling down the list of animations that are actually going to be played.

The priority layer illustrated in Figure 16 is choosing between the rider ground IK movements, G-forces on the rider and the rider's steering control. While it could conceivably build a large multi-axis blender for all of these animations, the amount of source data would multiply. Combining G-Forces and steering forces into a multi-axis blender node, for example, would likely require each G-Force an-

Figure 16



imation to be represented with each steering animation. This would total out at about 56 animations as opposed to the current 15. The initial concept of this tree included a fully blended system, but simply diagramming during the planning phase pointed out that the large number of animations would be a problem.

10 Memory Performance

The number of animations really snuck up on the team. The fidelity of this system comes at a cost. Each 'Mech has 152 animations which totals out to 3,600 animations in MechWarrior4 alone. This does not include all the expansion packs and Mercenaries, which double that number. The working set of actual blended animations comes out to about 360 animations per frame or 7,200 animated channels. Each IK animation requires five animations; flat, up, down, left and right. Just 30 animation states that have ground IK would equal a surprising 150 animations.

The memory required to hold that much data is significant, but fortunately, there is a solution for this. Each animation needs to be aggressively optimized, and each channel in the animation must have its own optimization as well. By doing this, the channels in the hierarchy that must be absolutely accurate can be optimized with tighter thresholds. The remaining channels can be opti-

mized at a lower fidelity. Care needs to be taken to not over-optimize, as well. At least one key frame for each channel is needed for successful blending. Just because a channel is not animating does not mean it does not have a position. If this key frame is not available, the blender will not be able to blend the channel with other channels.

It is critical to have a tool that can data-mine animation size. The MechWarrior engine provides an animation viewer that can open up lists of animations. This tool reports the total size of the animation set and can dig down into each animation. Inside each animation, the tool reports exact details of the animation, including key frame sizes, header sizes and the key frame data. It serves as an invaluable tool for finding optimization problems as well as for diagnosing exporter errors in the animation. Using optimization alone, we cut the size of animations per 'Mech are minimized from a few megs to about 700k. This 700k includes all 152 animations.

11 CPU Performance

Cache performance issues are, by far, the number one performance issue in the animation system. In order to minimize cache issues, do not iterate and interpolate animation that is insignificant. Calculating the blend tree and weights before iteration allows the user to completely skip unused branches. Secondly, keep the data small. The smaller the data, the faster it will load and process. Finally, keep key frame times separate from key frame data in the animation format. Iteration of the key frames can very often go through large sections of the animation. If the key frame format includes the key frame time and key frame data, all the key frame data is being loaded into the cache before necessary and likely throwing it away again before it is used. Keeping key frame time separate from channel data produces less cache misses and makes pre-fetching a little easier.

Blending systems can take a toll on the CPU processing as well. Large amounts of interpolation happen every frame, and if standard quaternion

slerp is being used, with four sin's, an arcos and a square root normalize, performance will suffer. An optimized interpolator and normalize function are essential. "Hacking Quaternions" [?] is a good source for information on this issue. As each animation system is going to behave differently, someone familiar with assembly optimization should review the animation code to suggest processor and cache improvements. Lots of little improvements can be preformed to speed-up this type of code.

12 Helping the Animator

There has been much discussion from a technical standpoint about establishing this type of system. But the most important part of the process is the animators. Blended IK is wholly dependant on animators. Programming teams must fully support the artists, rather than just leave them to "create their art". The MechWarrior programming staff wrote several plugins for the animators' art package. These tools range from exporters to fully integrated animation plug-ins that have very little game technology.

Figure 2: MechWarrior Running



All of the animations are exported from the art package, and very little is done procedurally to the 'Mechs during run-time. Some of the animation is possible to create procedurally, but this is performed within the art package instead of during runtime. BY not performing runtime procedural alterations of the animation, the artist is able to

fix errors in the procedural algorithms. If the position of a leg is procedurally calculated based on an IK chain, that chain might reach the target position in a strange and undesirable way. When these errors occur, the artist can adjust the animation to look correct. These modifications proved to be invaluable as more 'Mechs were designed with complex leg structures and as the art package was not always able to satisfactorily solve those IK. It takes much less time and effort for the artist to correct the errors than to fix the algorithm that produced the error. Allowing the artists to adjust things also allows them to modify an animation to be more aesthetically pleasing and further modify the procedural animation. The artist can put in subtle differences in the motion to make it look less procedural, or he can completely abandon the procedural animation all together and add something created from scratch. The artist can also choose to not modify the animation further.

Figure 3: MechWarrior Walking



Three main content generation tools were created for MechWarrior. The Mirror Tool, Reverse Tool and Hill Tool. The Mirror Tool can take any animation and mirror it on an axis. This allows the animators to quickly take a step forward left leg animation and turn it into a step forward right leg animation. The Reverse Tool allows animators to completely reverse an animation in time. Forward walks become backward walks. With a little modification, they alter the weight and timing of the animation and it looks convincingly different. The Hill Tool was by far the most complicated animation generation tool we produced. For the Hill

Tool, a plug-in was created that took an animation designed for flat ground and replicated it to uphill, downhill, lefthill and righthill animations. The artists had an array of inputs for the tool that would scale and adjust specific joints in the hierarchy to help them create a smooth animation. When run, the tool will re-adjust the IK handles of the feet to match the ground angles and adjust other joints with the data in the input arrays. The artists work first with a simple walk animation to determine the appropriate inputs needed, and once they have those inputs, they run the entire suite of flat ground animations through it to get the full list of hill animations.

Animation count was also reduced by sharing animations between 'Mechs at runtime. As 'Mechs are varied in size, shape and joint count, this can be relatively challenging and is not always possible. Where available, the artists are allowed to specify animations from another 'Mech to be used for the current 'Mech. Many commercially available packages also exist to facilitate off-line sharing of motion data between characters. Due to memory concerns this is the one exception of run time alteration.

Figure 4: MechWarrior Jumping



13 Conclusion

13.1 Blended IK and More

Generic IK systems have a hard time imparting the sense of timing and motion that an animator can produce, even when combined with complicated physics. Many times, looking real is not actually being real. Because of this, blending IK is a great method of having a character act and react to its environment. Punching, getting punched, kicking objects and picking up objects are all problems that are very hard to solve with math. Math has a hard time determining how a character distributes his weight when he picks up an object on a table. Maybe the character uses his other arm to hold his balance or maybe he bends at the knees. Granting the responsibility of this process to the animators can greatly increase the fidelity. Whether a sports game, fighting game, platform game or FPS, any game with animation can apply these methods. A blending system can breathe life into characters. Simply allowing animators to do their jobs makes our games infinitely better.

References

- [1] Ken Shoemake, *Animating Rotations with Quaternion Curves*, SIGGRAPH '85, pp. 245-254.
- [2] Jonathon Blow, *Hacking Quaternions*, Game Developer Magazine, March 2002.